Compositional Verification of Composite Byzantine Protocols

Qiyuan Zhao, George Pîrlea,
Karolina Grzeszkiewicz,
Seth Gilbert, Ilya Sergey

NUS National University of Singapore

For CCS 2024

Distributed Protocols

- Distributed systems are important!
  - Scalability, reliability, performance, …
  - Theoretical foundation: distributed protocols
    - Defining how a node collaborates with other nodes

It is well-known that distributed systems are very important these days.
They support various Internet services, and usually they can provide better scalability, reliability, performance than traditional centralized systems.

While the benefits of distributed systems are clear, fundamentally their correctness relies on the underlying distributed protocols, where a distributed protocol defines how a distributed computing **node**, will collaborate with other nodes to <u>solve a specific problem</u>.

Byzantine Fault Tolerance

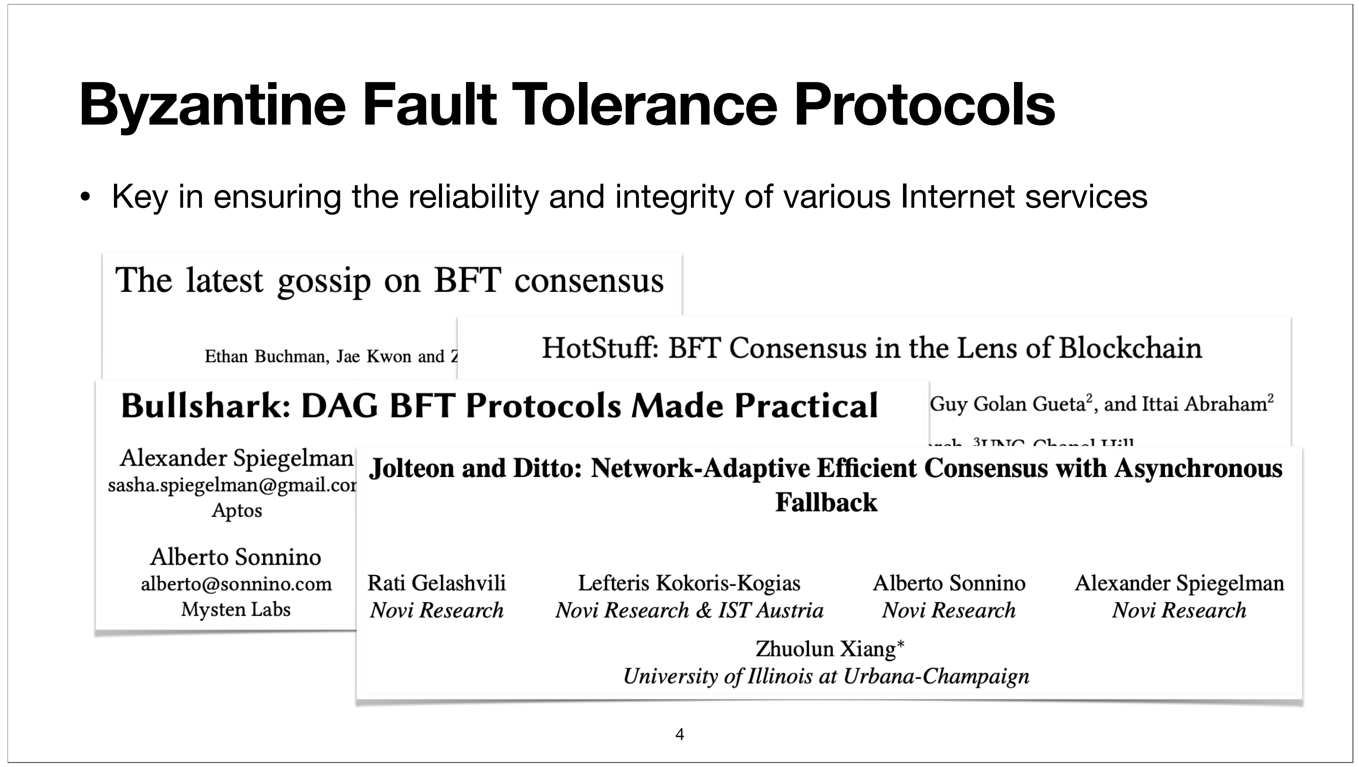
- Fault tolerance: a key goal in protocol design
- Byzantine fault:
  - Faulty nodes that can deviate from the protocol arbitrarily

The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

For a distributed protocol to be practical, it must account for potential faults in real-world systems, such as node crashes, message drops and (message) delays.
A fault-tolerant protocol can work correctly even in the presence of faults.

Among the various notions of faults, the Byzantine fault, which was initially introduced in this paper, has received particular attention.
A Byzantine node, meaning a node experiencing Byzantine fault, can deviate from the protocol arbitrarily.
Due to such characteristic, Byzantine nodes can represent malfunctioning nodes, or, even malicious attackers trying to corrupt the system.



To address the challenges posed by Byzantine faults, Byzantine fault tolerance protocols, or BFT protocols, have been developed for ensuring the reliability and integrity of security-critical systems such as blockchains.

The challenge of designing efficient BFT protocols has been a long-standing research problem, and in recent years, new BFT protocols continue to be proposed.

BFT Protocols Are Hard to Get Right

However, the designs of BFT protocols are often complicated and prone to bugs.
Some BFT protocols have been found to contain deep bugs in their design.

BFT Protocols Are Hard to Get Right

dranov / **protocol-bugs-list**

Errors found in distributed protocols

Protocol	Reference	Violation	Counter-example	#Year(s) taken to discover the bug
Sync HotStuff	[Abraham et al. 2019]	safety & liveness	[Momose and Cruz 2019]	≤ 1
Tendermint	[Buchman 2016]	liveness	[Cachin and Vukolić 2017]	≈ 1
hBFT	[Duan et al. 2015]	safety	[Shrestha et al. 2019]	≈ 4
Zyzzyva	[Kotla et al. 2007; Kotla et al. 2010]	safety	[Abraham et al. 2017]	≈ 7
FaB Paxos	[Martin and Alvisi 2005; Martin and Alvisi 2006]	liveness	[Abraham et al. 2017]	≈ 12
PBFT[1]	[Castro and Liskov 1999]	liveness	[Berger et al. 2021]	≈ 22

Source: https://github.com/dranov/protocol-bugs-list

To illustrate this, let me show you a list of bugs in various BFT protocols. This list is publicly available on Github.

As you can see, the bugs violate different aspects of guarantees, and the time it took to uncover them ranges from 1 year to over twenty years.

BFT Protocols Are Hard to Get Right

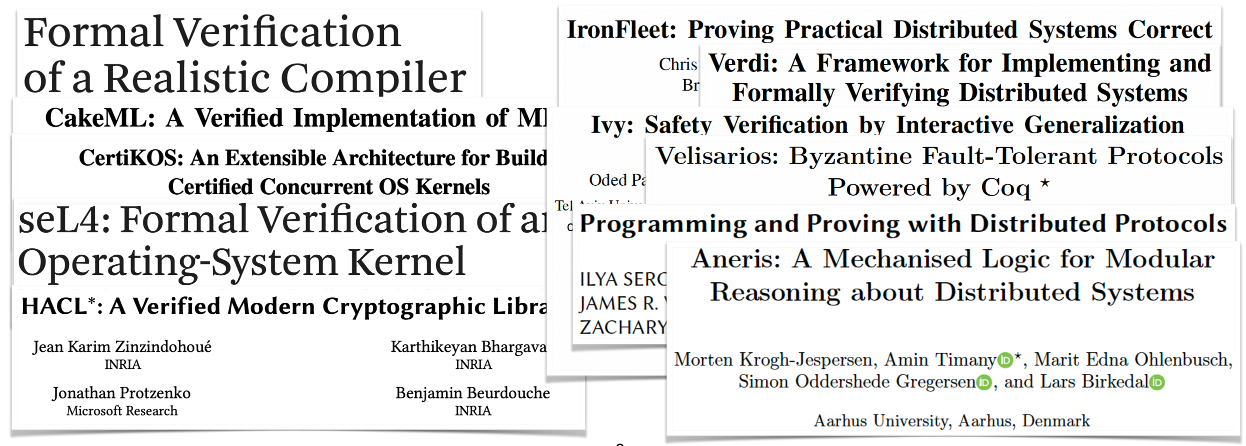
jepsen QUINT APALACHE

- Testing or model checking BFT protocols may not be effective
  - Byzantine behavior ⇒ large search space
  - Precisely capturing Byzantine behavior is difficult

Even though there are a bunch of tools for specifying, model checking or testing distributed systems, they may not be effective in exposing the bugs in BFT protocols.
One reason is that the non-determinism nature of Byzantine behavior leads to large search space, which can cause state explosion in model checking, and for testing, we need good heuristics to effectively sample testing scenarios.
Additionally, precisely capturing Byzantine nodes' behavior is also difficult! **For example, we often need to constrain** Byzantine nodes in a realistic setting, such as ensuring they cannot forge digital signatures. Formally expressing such constraints and applying them can be subtle.

Verification Builds Trust

- Reducing the risk of having bugs by formal verification
  - Proving properties rigorously with proofs aided/checked by machine



A promising way to reduce the risk of having bugs is to do formal verification by proving properties rigorously with proofs aided/checked by machine.
There have been many verification projects **targeting at large systems**, including compilers, operating systems, and cryptographic libraries.
There is also a thread on formally verifying distributed systems.

Verification is Also Laborious

IronFleet: Proving Practical Distributed Systems Correct

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch,
Bryan Parno, Michael L. Roberts, Srinath Setty, Brian Zill

Microsoft Research

“Proofs take 39253 LoC in total”

Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq *

Vincent Rahli ✉, Ivana Vukotic, Marcus Völp, Paulo Esteves-Verissimo

SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
firstname.lastname@uni.lu

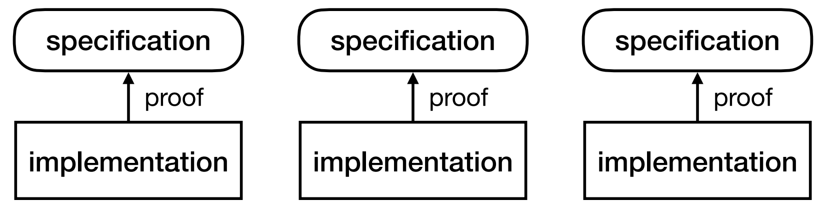
“Verifying PBFT takes around 20000 lines of specs and around 20000 lines of proofs”

- Such great efforts are difficult to reuse!

While verification builds trust, it is also laborious.
For example, the iron fleet project required around 40k lines of proofs **in total.**
The Velisarios project aimed at verifying the safety of the PBFT protocol and it took around 20000 lines of specs and around 20000 lines of proofs.
Although both projects **represent excellent efforts**, their frameworks do not **explicitly support reusing** verified protocols when verifying a new protocol.

Compositionality For The Win

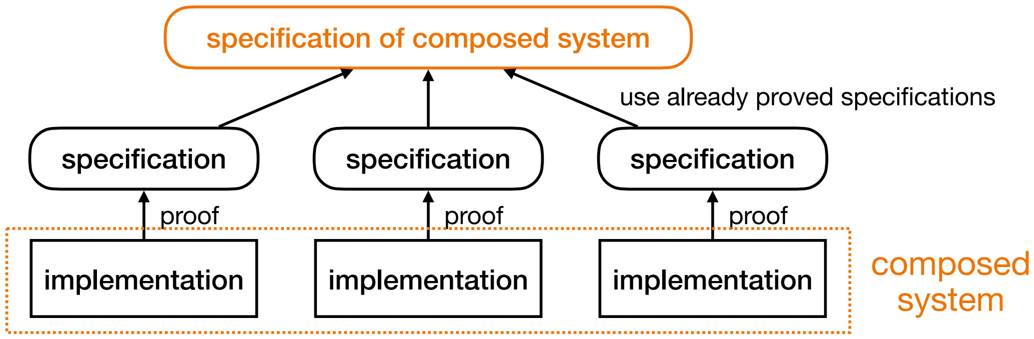
- Compositionality: the conventional wisdom in doing verification
  - Separation of specification and implementation
  - Modularity & proof reuse



In the verification community, the conventional approach to **achieving** proof reuse is through compositionality.
Compositionality **allows for** the separation of specification and implementation.
Moreover, compositionality **enables** modularity and proof reuse. Individual components can be verified separately,

Compositionality For The Win

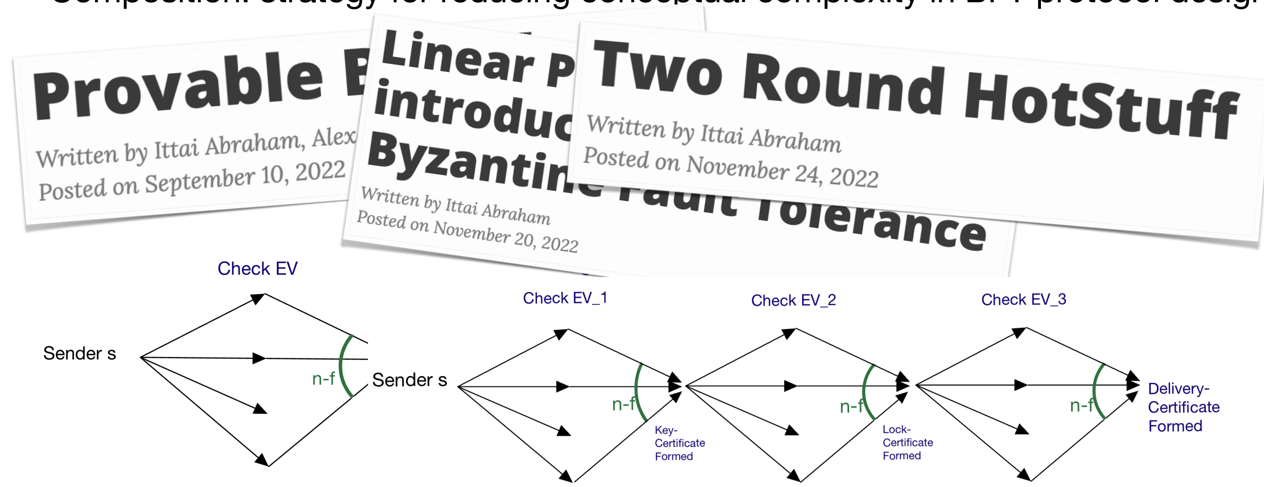
- Compositionality: the conventional wisdom in doing verification
  - Separation of specification and implementation
  - Modularity & proof reuse



Once these components are integrated into a larger system, we can **reuse the already proved specifications of those components** to derive the overall specification, **without the need to reprove everything from scratch**.

Compositionality For The Win

- Composition: strategy for reducing conceptual complexity in BFT protocol design



Blog source: https://decentralizedthoughts.github.io/
Image credit: https://decentralizedthoughts.github.io/2022-09-10-provable-broadcast/

On the other hand, composition is a strategy for reducing conceptual complexity in BFT protocol design.
There have been a series of blog posts on how to construct complex BFT protocols with some protocol as the building block.
The building block might be iterated for several times to strengthen its guarantee.

We want to make verification compositional for (potentially composite) BFT protocols.

To put it simply, what we want to achieve in this work is make verification compositional for (potentially composite) BFT protocols.

Our Contribution

- BYTHOS: streamlining the verification of BFT protocols and their compositions
  - Embedded in the Coq proof assistant ⇒ foundational
  - The first framework that supports:

    ☑ Reasoning about Byzantine faults

    ☑ Modular safety & liveness proofs of BFT protocols

    ☑ Proof reuse for verifying composite BFT protocols

    ☑ Executable reference implementation extracted to OCaml

To this end, we propose Bythos, the framework for streamlining the verification of BFT protocols and their compositions. Bythos is embedded in the Coq proof assistant. It provides foundational guarantee on the properties that we can prove using it. **To the best of our knowledge**, it is the first framework supporting the following points altogether: …

Our Contribution

- BYTHOS: streamlining the verification of BFT protocols and their compositions
  - Embedded in the Coq proof assistant ⇒ foundational
  - The first framework that supports:

    ☑ Reasoning about Byzantine faults

    ⭐ **Modular safety & liveness proofs of BFT protocols**

    ⭐ **Proof reuse for verifying composite BFT protocols**

    ☑ Executable reference implementation extracted to OCaml

Our technical **novelty** focuses on **the composition aspect**, specifically …

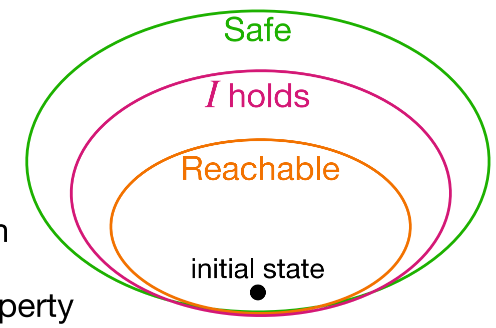
Outline

- Knowledge-Driven Proof Methodology
- Higher-Order Functor for Protocol Composition

Let me first introduce **our** knowledge-driven proof methodology, which is key to modularizing safety and liveness proofs.

Proving Safety Properties

- Safety: “bad thing never happens”
- The standard approach to proving safety:
  - Finding an inductive invariant I
    - Inductive: I is preserved after any transition
  - Showing that I implies the desired safety property



The safety properties of a distributed protocol **assert** that bad things will never happen **during the protocol execution**.
By modeling the distributed system as a state machine, proving safety amounts to showing that the set of reachable states is included in the set of safe states.
The standard approach to proving safety is to first find an inductive invariant I, which is preserved after any transition of the state machine, and then show that I implies the desired safety property.
Intuitively, the inductive invariant is for **approximating** the protocol, by (next slide)

Inductive Invariants

- Summarize the *knowledge* (or, causality) about protocol execution
  - “What we can know about the past by looking at the current state”
- Coming up with the inductive invariant all at once is difficult!

Planning for Change in a Formal Verification of the Raft Consensus Protocol

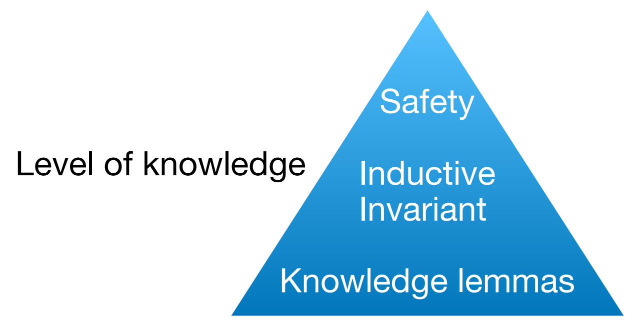
Doug Woos James R. Wilcox Steve Anton
Zachary Tatlock Michael D. Ernst Thomas Anderson
University of Washington, USA
{dwoos, jrw12, santon, ztatlock, mernst, tom}@cs.washington.edu

*“We present the first formal verification of state machine safety for the Raft consensus protocol. […]
This proof required iteratively discovering and proving* ***90*** *system invariants.”*

… summarizing the knowledge (or, causality) about protocol execution.
The knowledge or causality **here** can be basically stated in the form like “what we can know about the past by looking at the current state”.
However, coming up with the **useful** inductive invariant all at once is difficult.
For example, in the Verdi project where the safety of the Raft protocol was verified, in total 90 invariants were **involved** in the proof.

Knowledge-Driven Proof of Safety

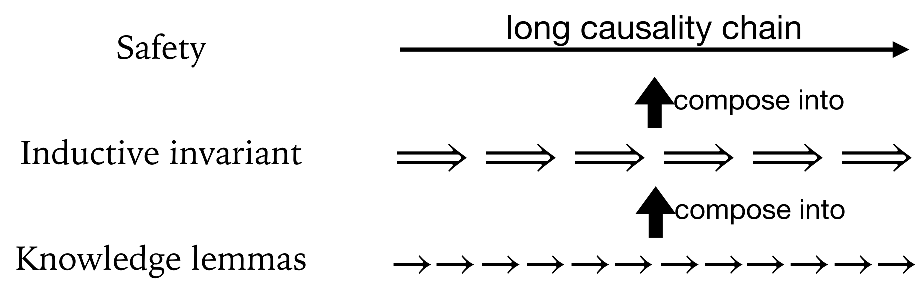
- **Knowledge lemmas**: Systematically capturing low-level properties of the protocol that ***directly*** follow from the protocol design
- Higher-level knowledge can be **incrementally** built upon lower-level knowledge



To **reduce the intellectual burden** of finding the inductive invariant, we propose **the concept of knowledge lemmas** for systematically capturing low-level properties of the protocol that directly follow from the protocol design.
From the perspective of knowledge, these lemmas **represent the kind of low-level knowledge that can be easily obtained by observing the protocol.**
The higher-level knowledge, including the inductive invariant and the safety, can be incrementally built upon lower-level knowledge.

Knowledge-Driven Proof of Safety

- **Knowledge lemmas**: Systematically capturing low-level properties of the protocol that ***directly*** follow from the protocol design
- Higher-level knowledge can be **incrementally** built upon lower-level knowledge



Here, we **might as well** think of safety as a long causality chain.
With inductive invariant based reasoning, we derive safety by **repeatedly applying** the inductive invariant, **where each arrow symbol indicates an application**. This is much like how we would compose smaller parts into a larger chain.
Similarly, the inductive invariant itself can be composed from knowledge lemmas, where each small arrow symbol indicates the application of **a specific knowledge lemma**.

So this picture illustrates how knowledge can be composed incrementally.
For the formal statements of knowledge lemmas and how they lead to modularity, you can check our paper.

Outline

- Knowledge-Driven Proof Methodology
- Higher-Order Functor for Protocol Composition

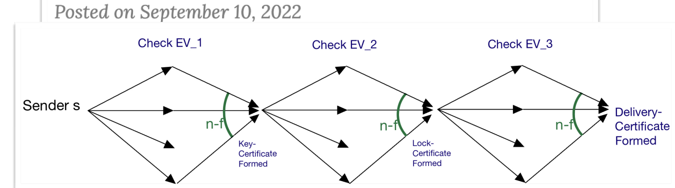
The second major technical contribution of our paper is related to composite protocols.

Sequential Composition of Protocols

- Sequential composition can help achieve stronger guarantee
  - Sequencing the same protocol for multiple times gives stronger guarantee
  - Composing a protocol with certain "protocol plugins" grants it new properties

Provable Broadcast

Written by Ittai Abraham, Alexander Spiegelman
Posted on September 10, 2022



As easy as ABC: Optimal (A)ccountable (B)yzantine (C)onsensus is easy!

Pierre Civit[1], Seth Gilbert[2], Vincent Gramoli[3,4], Rachid Guerraoui[4] and Jovan Komatovic[4]

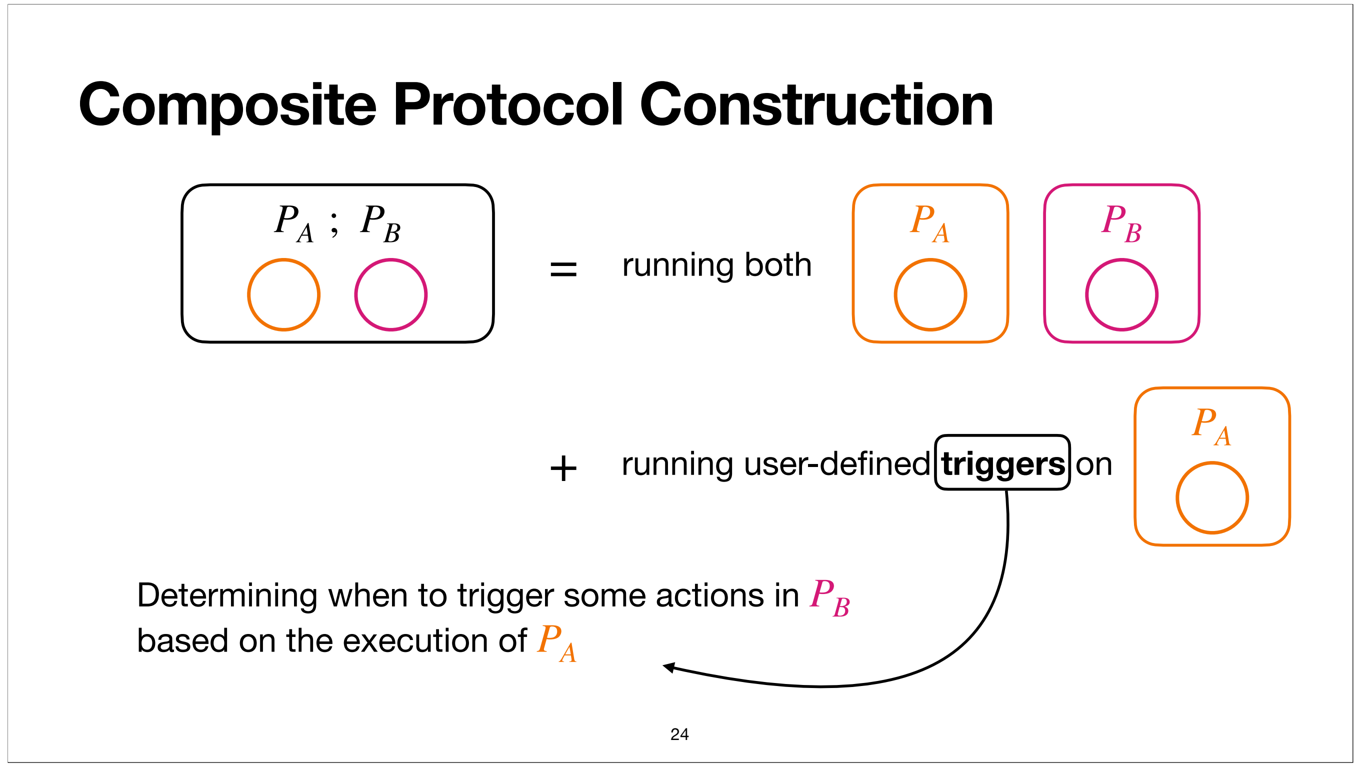
[1]Sorbonne University, CNRS, LIP6
[2]NUS Singapore
[3]University of Sydney
[4]EPFL

We identify that sequential composition is an important form of protocol composition, since it can allow a protocol to have stronger guarantee.
Sequencing the same protocol for multiple times gives stronger guarantee. This is illustrated in the blog post that I previously mentioned.
Additionally, composing a protocol with certain "protocol plugins" grants it new properties.
For example, this paper introduces such a protocol plugin that by running the protocol plugin after an arbitrary BFT consensus protocol, we can endow the consensus protocol the so-called accountability property, which is useful in certain cases.

Functor for Protocol Composition

- In BYTHOS, a protocol is encapsulated as a Coq module
- **Composition functor**: given two protocol modules, constructs a new one
  - Allows for composing multiple protocols

In Bythos, a protocol is encapsulated as a Coq module.
Sequentially composing protocols is enabled by **the composition functor, which** is basically a function over protocol modules: given two protocol modules, the functor produces their sequential composition.
The produced composite protocol is still a protocol module, so it can be plugged back into the functor to be composed with other protocols, which makes it possible to compose multiple protocol instances.



A node running the composite protocol “PB after PA” can be regarded as having two threads running PA and PB respectively.
In addition, the node runs the user-defined triggers, where trigger is the mechanism in our framework for …

Composing Proofs

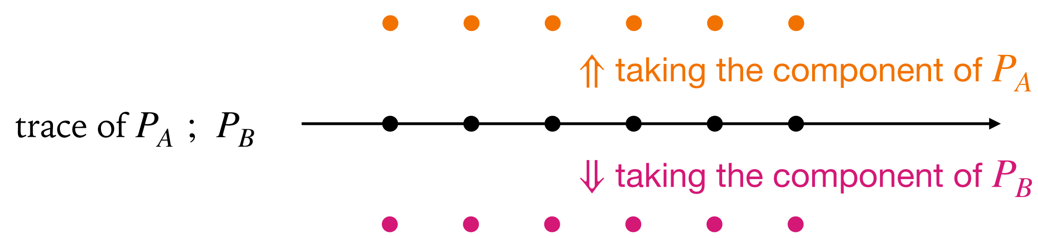
- The execution of a composite protocol can be projected into the executions of sub-protocols

trace of $P_A \; ; \; P_B$

From the way protocols are sequentially composed, an important observation is that …

Composing Proofs

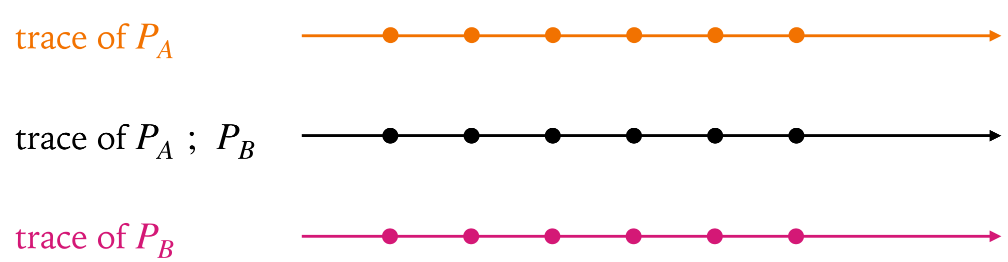
- The execution of a composite protocol can be projected into the executions of sub-protocols



In other words, we can think the trace of the composite protocol as a combination of the traces of PA and PB.

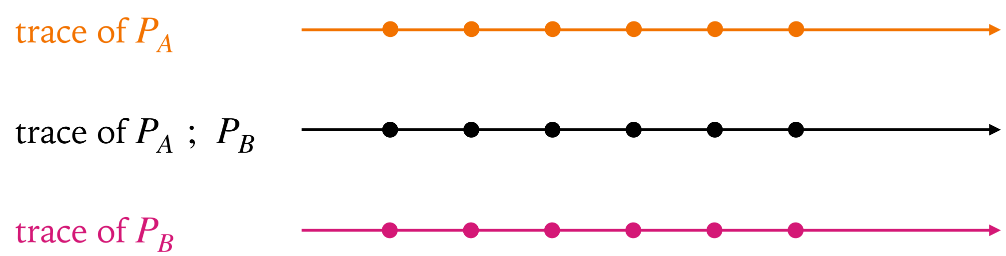
Composing Proofs

- The execution of a composite protocol can be projected into the executions of sub-protocols



Composing Proofs

- Allows for composing proofs of sub-protocols by **lifting**
  - Properties of P_A (P_B) should hold on the P_A (P_B) component of $P_A \ ; \ P_B$
  - Liveness properties can be even composed across different protocols!



Since safety and liveness properties are defined in terms of traces, this observation implies that we can **lift** the properties of PA and PB to the composite protocol. More precisely, …
And the story does not end here; for liveness properties, we can **not only** lift them, but also compose the liveness properties from different sub-protocols, **to derive the overall liveness of the composite protocol.**
Again, you can check our paper to see how this is possible in detail.

So, these are the two main technical contributions of our paper.

Proof Efforts

- Verified 3 asynchronous BFT protocols with their compositions
- BYTHOS + verified case studies: around 7100 lines of Coq code

Library	Component	Spec	Proof	Total
BYTHOS (Sec. 3)	System (Sec. 3.1)	729	465	1194
	Liveness (Sec. 3.2)	160	181	341
	Composition (Sec. 3.3)	329	255	584
	Utilities	184	157	341
	Total	1402	1058	2460
Provable Broadcast (Sec. 2)	Implementation (Sec. 2.1)	121	6	127
	Safety (Sec. 2.2)	404	320	724
	Liveness (Sec. 2.3)	92	67	159
	Composition (Sec. 2.4)	85	10†	95
	Total	702	403	1105

Reliable Broadcast (Sec. 4.1)	Implementation	130	6	136
	Safety (Sec. 4.1.1)	448	432	880
	Liveness (Sec. 4.1.2)	144	161	305
	Total	722	599	1321
Accountable Confirmer (Sec. 4.2)	Implementation	237	109	346
	Safety	619	709	1328
	Liveness (Sec. 4.2.2)	172	200	372
	Total	1028	1018	2046
Accountable Reliable Broadcast (Sec. 4.3)	Implementation	33	0	33
	Connector (Sec. 4.3.1)	48	92	140
	Liveness (Sec. 4.3.1)	3	7	10
	Total	84	99	183

In our paper, we verified 3 asynchronous BFT protocols with their compositions to **apply these techniques**.
In total, our framework and all verified case studies take around the 7100 lines of Coq code.

Summary

- BYTHOS: streamlining the verification of BFT protocols and their compositions
- Facilitating safety & liveness proofs with knowledge-driven proof methodology
- Allowing effective proof reuse in verifying composite BFT protocols

Github Repo

Paper

Thank you! 😃